Amélioration de la Scalabilité Mémoire du code GYSELA

Fabien Rozar

ComPAS'2014 : Parallélisme / Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Amélioration de la Scalabilité Mémoire du code GYSELA

Fabien Rozar

IRFM, CEA Cadarache, FR-13108 Saint-Paul-les-Durance
Maison de la simulation, CEA Saclay, FR-91191 Gif sur Yvette
fabien.rozar@cea.fr

Résumé
Les simulations gyrocinétiques nécessitent d'importants moyens de calcul. Jusqu'à présent, le code semi-Lagrangien GYSELA réalise des simulations sur quelques dizaines de milliers de cœurs de calcul (65k cœurs). Mais pour comprendre plus finement la nature de la turbulence des plasmas, nous devons raffiner la résolution de nos maillages, ce qui fera de GYSELA un candidat sérieux pour exploiter la puissance des futurs machines de type Exascale. Le fait d'avoir moins de mémoire par cœur est une des difficultés majeures des machines envisagées pour l'Exascale. Cet article porte sur la réduction du pic mémoire. Il présente aussi une approche pour comprendre le comportement mémoire d'une application utilisant de très grands maillages. Ceci nous permet d'extrapoler dès maintenant le comportement de GYSELA sur des configurations de type Exascale.

Mots-clés : Exascale, Scalabilité mémoire, Réduction de l'empreinte mémoire, Physique des plasmas.

---

1. Introduction

Depuis plusieurs années, la fréquences des processeurs n'augmente plus. Au lieu de doubler la vitesse des processeurs tous les 18-24 mois, le nombre de cœurs par nœud de calcul suit la même loi. Les architectures parallèles récentes présentent un modèle de mémoire hiérarchique et une tendance de ces machines est d'offrir de moins en moins de mémoire par cœur. Cette tendance est identifiée comme l'un des challenges pour l'utilisation de l'Exascale [13] et est l'une des raisons de cette étude.
Durant la dernière décennie, la simulation de la turbulence de plasmas de fusion dans les Tokamak a impliqué un nombre croissant de personnes venant des mathématiques appliquées et de la programmation parallèle [1]. Ce type d'application fait partie des candidats qui seront capables d'utiliser la première génération de machines Exascale. Le code GYSELA exploite déjà efficacement les capacités des super-calculateurs [9]. Dans cet article, nous nous intéressons en particulier à sa consommation mémoire. C'est un point crucial pour la simulation de cas physiques plus conséquents avec une quantité de mémoire toujours limitée par cœur.
Un module spécifique à l'application GYSELA a été développé pour permettre la génération d'une trace mémoire. Néanmoins, notre but final (pas encore atteint) est de définir une méthodologie et une bibliothèque portable pour aider le développeur à optimiser l'utilisation de la mémoire pour un certain type d'applications scientifiques parallèles.

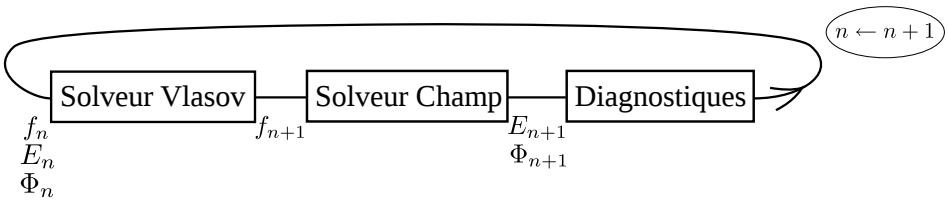

FIGURE 1 – Schéma numérique d'un pas de temps de GYSELA

Le but du travail présenté ici est de décomposer et de réduire l'empreinte mémoire de GYSELA pour améliorer sa scalabilité mémoire. Nous présentons un outil qui permet de visualiser les allocations/désallocations mémoire de GYSELA dans un mode hors-ligne. Un autre outil nous permet de prédire le pic mémoire en fonction de certains paramètres d'entrée. Cela est utile pour vérifier si les besoins mémoire futurs des simulations pourront tenir dans l'espace mémoire d'une machine donnée.
Cet article est organisé selon le plan suivant. La section 2 décrit succinctement le code GYSELA. La section 3 présente la consommation mémoire de GYSELA. La section 4 présente le module implémenté pour générer le fichier de traces des allocations/désallocations. Elle illustre aussi les capacités des outils de visualisation et de prédiction pour traiter les données du fichier de traces. La section 5 montre un exemple de réduction de l'empreinte mémoire et une étude de la scalabilité mémoire grâce à l'outil de prédiction. La section 6 conclut et présente nos perspectives.

2. Vue d'ensemble de GYSELA

Cette section donne une vue d'ensemble de l'algorithme global de GYSELA et introduit les principales structures de données utilisées.
GYSELA est un code global d'électrostatique non linéaire qui résoud les équations Vlasov-Maxwell gyrocinétiques. GYSELA est le couplage d'un solveur de Vlasov, qui modélise le mouvement des ions dans un tokamak, et d'un solveur de Maxwell qui calcule le champ électrostatique et permet d'appliquer une force sur les ions. L'équation de Vlasov est résolue grâce à une méthode semi-Lagrangienne [7] et l'équation de Maxwell est réduite à la résolution numérique d'une équation de type Poisson [8].
Dans ce modèle gyrocinétique, l'inconnue principale est une fonction de distribution f qui représente la densité des ions à une position donnée de l'espace des phases. L'exécution de GYSELA se décompose en une phase d'initialisation, des itérations temporelles et une phase de sortie. La figure 1 illustre le schéma numérique utilisé durant une itération temporelle. f_n représente la fonction de distribution, Φ_n le champ de potentiel et E_n le champ électrique qui correspond à la dérivée de Φ_n. L'étape Vlasov réalise l'évolution de f_n sur un pas de temps et l'étape du solveur champ calcule E_n. Périodiquement, GYSELA exécute des diagnostiques afin d'extraire des quantités physiques à partir de f_n, E_n et sauvegarde les résultats dans des fichiers HDF5 [5].
La fonction de distribution f agit sur une variable à 5 dimensions qui évolue avec le temps. Les 3 premières dimensions sont des dimensions d'espace, $\mathbf{x}_G = (r, \theta, \varphi)$ avec r et θ les coordonnées polaires dans la coupe poloïdale du tore alors que φ désigne l'angle toroïdale. Les deux dernières coordonnées modélisent l'espace des vitesses, $v_{\|}$ la vitesse le long d'une ligne de champ magnétique et μ le moment magnétique.
Soient N_r, N_θ, N_φ, $N_{v_{\|}}$ respectivement le nombre de points dans chaque dimension r, θ, φ, $v_{\|}$.

Dans le solveur de Vlasov, chaque valeur de μ est associée à un ensemble de processus MPI (un communicateur MPI). Dans chaque ensemble, une décomposition en domaine 2D nous permet d'attribuer à chaque processus MPI un sous-domaine pris dans les dimensions (r, θ). Donc un processus MPI est responsable du stockage du sous-domaine défini par $f(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_{\parallel} = *$ [1] $, \mu = \mu_{value})$. La décomposition parallèle est caractérisée par les valeurs $i_{start}, i_{end}, j_{start}, j_{end}, \mu_{value}$ qui sont connues localement. Ces domaines 2D découlent d'une décomposition par bloc classique de la direction r en p_r sous-domaines, et de la direction θ en p_θ sous-domaines. Le nombre de processus MPI utilisés durant une exécution est égal à $p_r \times p_\theta \times N_\mu$. Le paradigme OPENMP est aussi utilisé au sein de chaque processus MPI (#T threads dans chaque processus MPI) pour exploiter le parallélisme à grain-fin.

3. Goulot d'étranglement mémoire

3.1. Analyse

Une exécution de GYSELA nécessite une quantité de mémoire importante. Durant une exécution de GYSELA, chaque processus MPI est associé avec une valeur de μ (section 2) et manipule la fonction de distribution comme un tableau 4D et le champ électrique comme un tableau 3D. Le reste de la consommation mémoire vient en grande partie des tableaux utilisés pour stocker des valeurs précalculées, des buffers MPI pour concaténer des données à l'envoi ou à la réception et des buffers utilisateurs OPENMP pour calculer des résultats temporaires. Quasiment tous ces tableaux sont alloués durant l'initialisation de GYSELA.

Afin de mieux comprendre le comportement mémoire de GYSELA, on enregistre à chaque allocation (au niveau des instructions "allocate") : le nom du tableau, son type et sa taille. En utilisant ces données nous avons réalisé un *strong scaling* illustré par le Tableau 1 (16 threads par processus MPI). D'un point de vue de la mémoire, l'étude de ce *strong scaling* consiste à exécuter le programme avec un maillage assez grand et à mesurer la quantité mémoire consommée par processus en augmentant au fur et à mesure le nombre de processus MPI utilisés pour la simulation. Si pour une simulation donnée avec n processus on utilise x Giga octets de mémoire par processus, on peut espérer dans le cas idéal qu'en faisant la même simulation avec $2n$ processus on obtienne une consommation mémoire de $\frac{x}{2}$ Giga octets. Dans ce cas, on dit que la *scalabilité* mémoire est parfaite. Mais en pratique, ce n'est généralement pas le cas à cause des surcoûts mémoire dus à la parallélisation.

Le Tableau 1 montre l'évolution de la consommation mémoire en fonction du nombre de cœurs et de processus MPI. Le pourcentage de la consommation mémoire par rapport au total de la mémoire consommée est donné pour chaque type de structure de données. Les dimensions du maillage sont les suivantes : $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$. Ce maillage est plus conséquent que ceux utilisés en production actuellement, mais il correspond aux besoins futurs, spécialement à ceux de la physique multi-espèces. Le dernier cas avec 2048 processus requiert 67.5 GB de mémoire par processus MPI. Nous associons habituellement un processus MPI à nœud de calcul. On peut remarquer que la mémoire requise est supérieure au 64 GB d'un nœud d'Helios [2] ou encore au 16 GB d'un nœud d'une Blue Gene/Q. Le Tableau 1 illustre aussi le fait que les structures 2D et les structures 1D ne bénéficient pas du parallélisme spatial induit par la décomposition de domaine. En fait, le coût mémoire des structures 2D ne dépend pas du tout du nombre de processus, mais plutôt de la taille du maillage et du nombre de threads. Dans le dernier cas à 32k cœurs, le coût des structures 2D est le principal goulot d'étranglement. Il prend 49 % de l'empreinte mémoire totale.

1. La notation $*$ représente toutes les valeurs d'une dimension donnée
2. http ://www.top500.org/system/177449

TABLE 1 – Strong scaling : taille des allocations statiques en Go (par processus MPI) et pourcentage de chaque type de donnée par rapport au total alloué

Nombre de cœurs Nombre de processus MPI	2k 128	4k 256	8k 512	16k 1024	32k 2048
structures 4D	209.2 67.1 %	107.1 59.6 %	56.5 49.5 %	28.4 34.2 %	14.4 21.3 %
structures 3D	62.7 20.1 %	36.0 20.0 %	22.6 19.8 %	19.7 23.7 %	18.3 27.1 %
structures 2D	33.1 10.6 %	33.1 18.4 %	33.1 28.9 %	33.1 39.9 %	33.1 49.0 %
structures 1D	6.6 2.1 %	3.4 1.9 %	2.0 1.7 %	1.7 2.0 %	1.6 2.3 %
Total par processus MPI en Goctets	311.5	179.6	114.2	83.0	67.5

Dans GYSELA, le surcoût mémoire pour les simulations sur un grand nombre de cœurs s'explique pour diverses raisons. De la mémoire supplémentaire est nécessaire par exemple pour stocker des coefficients durant une phase d'interpolation (pour le solveur Semi-Lagrangien de l'équation de Vlasov). Les buffers MPI constituent aussi des surcoûts mémoire. Les appels aux routines MPI impliquent souvent une copie des données qu'on veut envoyer ou recevoir dans le format approprié. Nous avons réduit certains de ces surcoûts mémoire. Cela a amélioré la scalabilité mémoire et nous a permis d'exécuter de plus grands cas physiques.

3.2. Approche

Il y a au moins deux approches pour réduire l'empreinte mémoire par processus d'une application parallèle. Premièrement on peut augmenter le nombre de nœuds utilisés pour la simulation. La taille des structures qui bénéficient de la décomposition de domaine diminue lorsque le nombre de processus MPI augmente. Deuxièmement, on peut gérer plus finement les allocations des tableaux afin de réduire spécifiquement les coûts mémoire qui ne passent pas l'échelle avec le nombre de threads/processus MPI et aussi limiter leur impact sur le pic mémoire.

Pour parvenir à réduire l'empreinte mémoire et pour repousser le goulot d'étranglement mémoire, nous allons ici nous concentrer sur la seconde approche.

Dans la version originale du code, la majeure partie des variables volumineuses est allouée durant la phase d'initialisation. Cette approche est justifiée pour les structures qui sont des *variables persistantes* en opposition aux *variables temporaires* qui peuvent être allouées dynamiquement. Dans cette configuration on peut, tout d'abord, déterminer rapidement l'espace mémoire requis sans exécuter complètement la simulation (la phase d'initialisation doit être uniquement exécutée). Cela permet à l'utilisateur de savoir rapidement si le cas "tient en mémoire" ou pas. Deuxièmement, cela évite les surcoûts en temps d'exécution dûs à la gestion dynamique des allocations. Mais un des désavantages de cette approche est que les variables utilisées localement dans certaines subroutines consomment leur espace mémoire durant toute l'exécution de la simulation. Comme l'espace mémoire devient un point critique quand un grand nombre de cœurs est utilisé, nous avons alloué une grande partie de ces variables temporaires avec des allocations dynamiques. Ceci a réduit le pic mémoire avec un impacte négligeable sur le temps d'exécution. Néanmoins, un des inconvénients des allocations dynamiques est que nous perdons les deux avantages des allocations statiques, et en particulier la possibilité de déterminer rapidement l'espace mémoire requis pour exécuter une simulation.

4. Outils de modélisation et de trace

Pour suivre la consommation mémoire de GYSELA et pour mesurer la réduction de l'empreinte mémoire, des outils complémentaires ont été développés : un module FORTRAN pour générer un fichier de traces des allocations/désallocations, et un script PYTHON de visualisation + prédiction qui exploite ce fichier trace. Les informations collectées de l'exécution de GYSELA grâce au module d'instrumentation est un composant clé de notre analyse mémoire. Certains aspects de l'implémentation de ces outils sont détaillés dans les sections suivantes.

4.1. Fichier de traces

GYSELA fait intervenir différents types de strutures de données et afin de gérer leurs allocations/désallocations, un module FORTRAN dédié a été développé pour les enregistrer dans un fichier : la *trace mémoire dynamique*. Puisque les processus MPI ont presque tous le même historique d'allocation/désallocation, nous produisons dans l'implémentation actuelle un seul fichier de traces pour les allocations/désallocations du processus MPI 0. Le travail restitué dans cet article est en partie financié par le projet NUFUSE G8-EXASCALE [3].

4.1.1. Etat de l'art.

Dans la communauté des chercheurs travaillant sur les outils d'analyse de performances d'applications parallèles, différentes approches existent. Souvent ces approches s'appuient sur des *fichiers de traces*. Un fichier de traces collecte des informations de l'application qui représentent un aspect de son exécution : le temps d'exécution, le nombre de messages MPI envoyés, le temps inoccupé (idle), la consommation mémoire, et plus encore. Mais pour obtenir ces informations, l'application doit être instrumentée. L'instrumentation peut être faite à 4 niveaux : dans le code source, à la compilation, à l'édition de lien ou durant l'exécution (*just in time*).
L'outil de performance SCALASCA [6] est capable d'instrumenter le code source à la compilation. Cette approche a l'avantage de couvrir toutes les parties du code et elle permet la personnalisation des informations à récupérer. Cette approche systématique donne une trace complète mais la récupération d'informations dans toutes les subroutines du code peut induire un surcoût en exécution important. Aussi avec une instrumentation automatique, la récupération de l'expression d'une allocation peut être difficile, ce qui se révèle nécessaire dans notre approche (cf. section suivante). L'ensemble d'outils EZTRACE [2] offre la possibilité d'intercepter les appels à un ensemble de fonctions. Cet outil peut rapidement instrumenter une application grâce à un lien avec des bibliothèques de tiers partie lors de l'édition de lien. Contrairement à notre approche, celle-ci ne nécessite pas une instrumentation du code, mais nous ne pouvons pas espérer récupérer l'expression utilisée lors de l'allocation avec cette approche. Les outils PIN [10], DYNAMORIO [3] ou MAQAO [4] produisent une instrumentation durant l'exécution. L'avantage est ici l'aspect générique de la méthode. Tous les programmes peuvent être instrumentés de cette façon, mais contrairement à notre approche, l'instrumentation dynamique introduit souvent un surcoût assez important en temps d'exécution.
Un autre logiciel couramment utilisé pour détecter les fuites mémoires est VALGRIND [11]. Ce logiciel utilise une instrumentation lourde au niveau binaire. Cette instrumentation lui permet de récupérer l'ensemble des accès mémoires effectué lors de l'exécution d'un programme. Parmi les outils développés à partir de VALGRIND, l'outil MASSIF permet de représenter graphiquement la consommation mémoire d'un programme. La Fig. 2 présenté dans la section 4.2 rempli le même objectif, à savoir avoir une vision d'ensemble de la consommation mémoire et situer le pic mémoire. Néanmoins, le réel désavantage de l'instrumentation binaire est le sur-

3. http ://www.nu-fuse.com

coût par rapport au temps d'exécution qu'il introduit, comme il l'a déjà était mentionné plus haut.
L'instrumentation binaire s'appuie sur un programme déjà compilé. Après compilation, il est difficile de faire le rapprochement entre les valeurs que contient la pile d'exécution lors d'une allocation et les variables du code source responsables de cette allocation. C'est un aspect qui est essentiel pour notre outil de prédiction de la consommation mémoire. Le but de l'outil de prédiction utilisé dans GYSELA est de prédire la consommation mémoire du programme si la valeur de certains paramètres change. La prédiction de la consommation mémoire est inaccessible si on ne peut pas faire le lien entre la taille des structures allouées et le nom des variables qui donnent les dimensions des allocations.
L'outil que nous avons développé nous permet de mesurer les performances de GYSELA, du point de vue mémoire. Un outil de visualisation a été développé pour permettre l'analyse de ce fichier de traces. Il offre une vision globale de la consommation mémoire et une vision précise autour du pic mémoire pour aider le développeur à réduire l'empreinte mémoire. La sortie sur terminal du script de post-traitement donne des informations précises au sujet des tableaux alloués au moment du pic mémoire. Etant donné un fichier de traces, il est possible d'extrapoler la consommation mémoire en fonction des paramètres d'entrée. Ce nouvel outil nous permet d'investiguer la scalabilité mémoire. Nous ne connaissons pas d'outil équivalent pour modéliser le comportement mémoire dans la communauté du HPC.

4.1.2. Implémentation.

Un module FORTRAN dédié à l'instrumentation a été développé. Il générera le fichier de traces. Nous pratiquons ensuite une analyse post-mortem sur celui-ci. Le module d'instrumentation propose une interface, *take* et *drop*, qui enveloppe les appels à `allocate` et `deallocate`. Les subroutines `take` et `drop` réalisent respectivement l'allocation et la désallocation d'un tableau manipulé et elles enregistrent l'action mémoire dans le fichier de traces.
Pour chaque allocation et désallocation, le module enregistre le nom du tableau, son type, sa taille et l'expression du nombre d'éléments. L'expression est nécessaire pour pouvoir faire de la prédiction. Par exemple, l'expression associée à cette allocation :

```
integer, dimension(:,:), pointer  :: array
integer                           :: a0, a1, b0, b1
... (intialisation des variables a0, a1, b0, b1)
allocate(array(a0:a1, b0:b1))
```

est

$$(a1 - a0 + 1) \times (b1 - b0 + 1). \tag{1}$$

Pour être capable d'évaluer ces expressions d'allocation en dehors de l'application, les variables qui y interviennent doivent être renseignées. Soit la valeur de la variable est fixée ou soit une expression arithmétique dépendant d'autres variables est renseignée. Ces actions sont réalisées respectivement par les subroutines *write_param* et *write_expr*. L'écriture des expressions permet de restituer les liens qui existent entre les différents paramètres. Ils sont essentiels pour que la prédiction soit correcte (cf. section 4.3). L'extrait de code suivant est un exemple de l'enregistrement des paramètres $a0, a1, b0, b1$:

```
call write_param('a0', 1); call write_param('a1', 10)
call write_param('b0', 1); call write_expr('b1', '2*(a1-a0+1)')
```

Pour reconstituer un historique plus précis des allocations mémoire, l'entrée/la sortie de subroutines que nous avons sélectionnées est enregistrée par l'interface *write_begin_sub* et

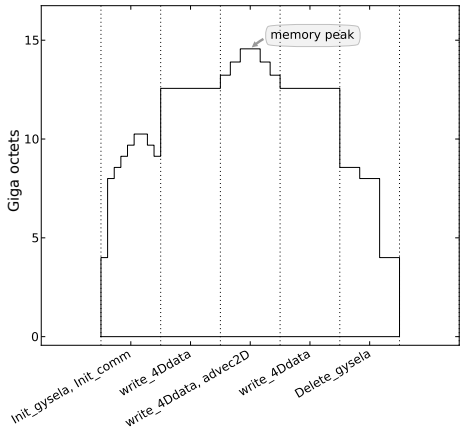

FIGURE 2 – L'évolution de la consommation mémoire dynamique durant une exécution très courte de GYSELA

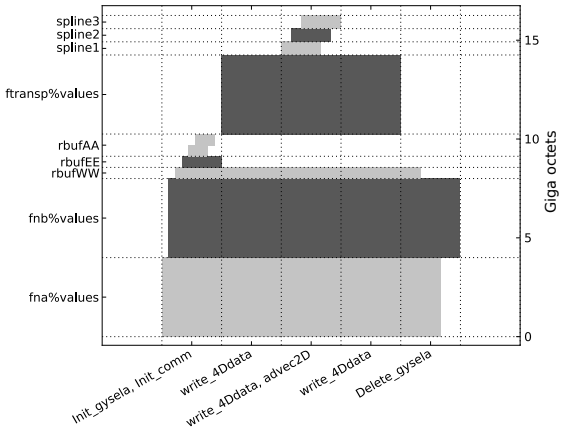

FIGURE 3 – Allocations et désallocations des tableaux utilisés dans diverses subroutines de GYSELA

write_end_sub. Ceci nous permet de localiser où se produisent les allocations/désallocations, ce qui est un aspect essentiel pour l'étape de visualisation.

4.2. Visualisation

Afin d'analyser finement la consommation mémoire, notre outil permet d'identifier les parties du code où l'usage de la mémoire atteint son pic. Le fichier de traces peut être volumineux, plusieurs Méga octets. Un script PYTHON a été développé pour visualiser ces traces. Cet outil aide le développeur à comprendre le coût mémoire des algorithmes qu'il manipule et lui donne des indices sur comment et où il peut appliquer des modifications pour diminuer l'empreinte mémoire. Ces informations sont restituées par deux types de graphiques.
La figure 2 trace la consommation de mémoire *dynamique* en Go en fonction du temps. L'axe des abscisses représente les entrées/sorties chronologiques des subroutines instrumentées. L'axe des ordonnées donne la consommation en Go. La figure 3 montre dans quelles subroutines les tableaux sont alloués. L'axe des abscisses reste identique au graphique précédent et l'axe des ordonnées montre les noms des tableaux. L'allocation d'un tableau correspond à un rectangle en gris clair ou en gris foncé dans sa ligne correspondante. La largeur des rectangles dépend de la durée de l'allocation.
Sur la figure 2, on peut localiser dans quelle subroutine le pic mémoire est atteint. Sur la figure 3, on peut ensuite identifier les tableaux qui sont effectivement alloués quand le pic mémoire est atteint. Grâce à ces informations, nous savons exactement où modifier le code pour réduire le pic mémoire.

4.3. Prédiction

Pour anticiper nos besoins mémoire avant de lancer une simulation donnée, il est utile de prédire la consommation mémoire pour un jeu de paramètres d'entrée. Grâce aux expressions des tailles de tableau et la valeur ou l'expression des paramètres numériques contenue dans le fichier trace, nous pouvons modéliser le comportement mémoire hors ligne. L'idée ici est de reproduire les allocations avec n'importe quel ensemble de paramètres d'entrée.
Parfois, la valeur d'un paramètre ne peut pas être exprimée en une ligne d'expression arithmé-

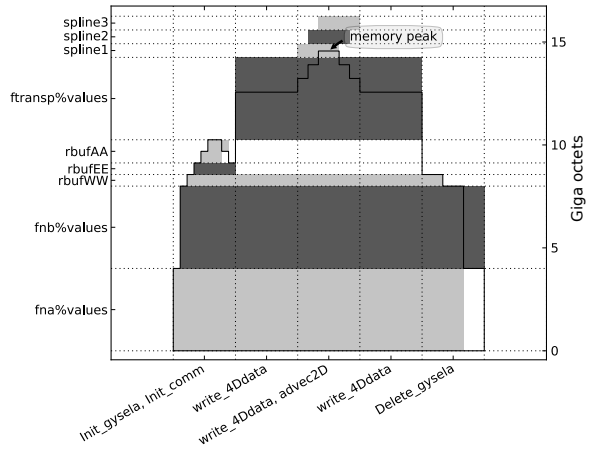

FIGURE 4 – Première visualisation de trace

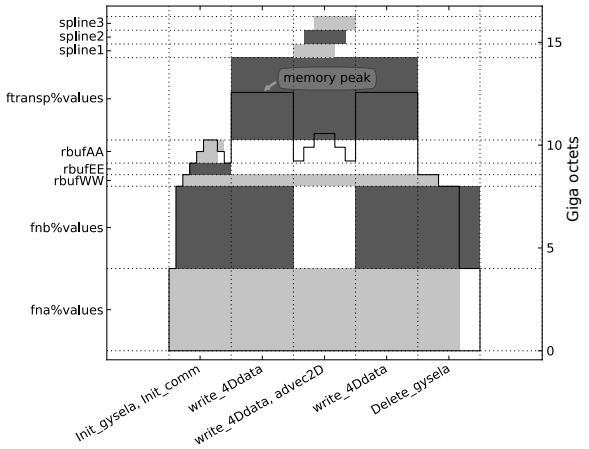

FIGURE 5 – Visualisation après optimisation

tique (e.g. une boucle d'optimisation multicritère pour déterminer cette valeur). Pour gérer ce cas, la partie de code FORTRAN qui retourne la valeur dans GYSELA est appelée depuis le script PYTHON. Cela est possible grâce à une compilation du code FORTRAN avec `f2py` [12].
En changeant la valeur d'un paramètre d'entrée, notre outil de prédiction PYTHON offre la possibilité d'extrapoler la consommation mémoire de GYSELA sur de plus grands maillages ou sur des configurations de super-calculateurs qui n'existent pas encore, e.g. pour faire une projection à l'Exascale. Les résultats de cet outil sont présentés dans la section 5.2.

5. Résultats expérimentaux

5.1. Réduction de l'empreinte mémoire

Réduire l'empreinte mémoire équivaut à diminuer le pic mémoire. Les figures 4 et 5 montrent l'impact sur la mémoire de quelques modifications du code. Après analyse du code, nous avons remarqué qu'au moment du pic mémoire, la structure transposée `ftransp%values` et la fonction de distribution `fnb%values` contiennent les mêmes données organisées différemment. Nous obtenons le fichier de traces de la figure 5 en désallouant `fnb%values` au moment du pic mémoire. Cette seule optimisation nous a permis de réduire le pic mémoire de 10% sur certaines simulations avec 32k cœurs. Ce type d'optimisation réduit efficacement le pic mémoire mais détériore un peu la lisibilité du code.
Grâce à cet outil, on peut voir qu'en fonction de la taille du maillage et du nombre de processus MPI et de threads OPENMP, le pic mémoire n'est pas toujours localisé au même moment. Ce comportement peut être expliqué par la dépendance entre la taille de certains tableaux caractéristiques et la valeur de certains paramètres d'entrée. Par exemple, la taille des buffers MPI est sensible aux paramètres de parallélisation. Dans GYSELA, les tailles des buffers temporaires sont sensibles au nombre de points dans les directions `r` et `theta`.
L'outil de visualisation donne un nouveau point de vue sur le code source. Cet outil nous a aidé à réduire en plusieurs passes les surcoûts mémoire et donc à améliorer la scalabilité mémoire du code GYSELA.

5.2. Prédiction sur des grands maillages

5.2.1. Scalabilité

Le tableau 2 présente le test en strong scaling avec les nouvelles allocations dynamiques et plusieurs améliorations algorithmiques que nous avons faites grâce à l'outil de visualisation

(non détaillées ici). L'outil de prédiction nous permet de reproduire le tableau 1 sur le même maillage, i.e. $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{\nu_\parallel} = 128$, $N_\mu = 2$.

TABLE 2 – Strong scaling : taille des allocations mémoires et pourcentage par rapport au total de chaque type de donnée au pic mémoire

Nombre de cœurs Nombre de processus MPI	2k 128	4k 256	8k 512	16k 1024	32k 2048
structures 4D	207.2 79.2%	104.4 71.5%	53.7 65.6%	27.3 52.2%	14.4 42.0%
structures 3D	42.0 16.1%	31.1 21.3%	18.6 22.7%	15.9 30.4%	11.0 32.1%
structures 2D	7.1 2.7%	7.1 4.9%	7.1 8.7%	7.1 13.6%	7.1 20.8%
structures 1D	5.2 2.0%	3.3 2.3%	2.4 3.0%	2.0 3.8%	1.7 5.1%
Total par processus MPI en Goctets	261.5	145.9	81.9	52.3	34.3

Le tableau 2 donne la consommation mémoire au pic mémoire. Pour obtenir ce tableau, la taille du maillage est constante et entre chaque colonne le nombre de processus MPI est doublé. Comme on peut le voir, sur le plus gros cas (32k cœurs), la consommation des structures 2D a été réduite à 20.8%. Aussi le gain mémoire sur ce cas est de **50.8%** sur la consommation globale en comparaison à ce indiqué dans le tableau 1. Les structures 4D contiennent les données les plus pertinentes utilisées durant le calcul et elles consomment la plus grande partie de la mémoire comme on la souhaitait naturellement. Les surcoûts mémoires ont donc été globalement réduits ce qui améliore la scalabilité mémoire de GYSELA et permet l'exécution de plus grandes simulations.

5.2.2. Investigation

En utilisant l'outil de prédiction, on peut investiger des maillages plus grands et la taille des machines requises pour manipuler cette quantité de données peut être estimée. Avec l'implémentation actuelle, pour mettre en œuvre le maillage $N_r = 2048$, $N_\theta = 4096$, $N_\varphi = 2048$, $N_{\nu_\parallel} = 256$, $N_\mu = 2$, le nombre de cœurs nécessaires serait de 524k cœurs, avec 64 Go par processus et 16 threads par processus.

6. Conclusion

La scalabilité mémoire du code GYSELA a été améliorée grâce à une gestion dynamique des allocations/désallocations. Une réduction de **50.8%** du pic mémoire a été réalisée sur certaines simulations à 32k cœurs.

Cet article décrit un module de modélisation et de trace mémoire et quelques outils de post-traitement qui permettent de réduire le pic mémoire. Avec ce jeu d'outils, la modélisation du comportement de l'empreinte mémoire au cours du temps de GYSELA est accessible. L'outil de prédiction permet d'extrapoler la consommation mémoire pour différents jeux de paramètres d'entrées en mode hors-ligne ; cet aspect est important aussi bien pour les utilisateurs finaux qui ont besoin de plus grandes résolutions ou de fonctionnalités gourmandes en mémoire, et

pour les développeurs qui ajustent leurs algorithmes pour une machine spécifique, e.g. de type Exascale.
Notre prochain objectif est d'implémenter une bibliothèque utilisable en C/FORTRAN. Le travail présenté dans cette article est la première étape de la construction d'une méthodologie qui aide les développeurs à améliorer la scalabilité mémoire de leur application parallèle.

Bibliographie

1. Åström (J.), Carter (A.), Hetherington (J.), Ioakimidis (K.), Lindahl (E.), Mozdzynski (G.), Nash (R. W.), Schlatter (P.), Signell (A.) et Westerholm (J.). – Preparing scientific application software for exascale computing. *In : Applied Parallel and Scientific Computing*, pp. 27–42. – Springer, 2013.
2. Aulagnon (C.), Martin-Guillerez (D.), Rué (F.) et Trahay (F.). – Runtime function instrumentation with eztrace. *In : Euro-Par 2012 : Parallel Processing Workshops*. Springer, pp. 395–403.
3. Bruening (D.), Garnett (T.) et Amarasinghe (S.). – An infrastructure for adaptive dynamic optimization. *In : [CGO 2003]*. IEEE, pp. 265–275.
4. Djoudi (L.), Barthou (D.), Carribault (P.), Lemuet (C.), Acquaviva (J.-T.), Jalby (W.) et al. – Maqao : Modular assembler quality analyzer and optimizer for itanium 2. *In : The 4th Workshop on EPIC architectures and compiler technology, San Jose*.
5. Folk (M.), Cheng (A.) et Yates (K.). – Hdf5 : A file format and i/o library for high performance computing applications. *In : Proceedings of Supercomputing*.
6. Geimer (M.), Wolf (F.), Wylie (B. J.), Ábrahám (E.), Becker (D.) et Mohr (B.). – The scalasca performance toolset architecture. *CCPE*, vol. 22, n6, 2010, pp. 702–719.
7. Grandgirard (V.), Sarazin (Y.), Garbet (X.), Dif-Pradalier (G.), Ghendrih (P.), Crouseilles (N.), Latu (G.), Sonnendrucker (E.), Besse (N.) et Bertrand (P.). – Computing ITG turbulence with a full-f semi-Lagrangian code. *Communications in Nonlinear Science and Num. Sim.*, vol. 13, n 1, 2008, pp. 81 – 87.
8. Hahm (T. S.). – Nonlinear gyrokinetic equations for tokamak microturbulence. *Physics of Fluids*, vol. 31, n9, 1988, pp. 2670–2673.
9. Latu (G.), Grandgirard (V.), Crouseilles (N.) et Dif-Pradalier (G.). – Scalable quasineutral solver for gyrokinetic simulation. *In : PPAM (2)*. pp. 221–231. – Springer.
10. Luk (C.-K.), Cohn (R.), Muth (R.), Patil (H.), Klauser (A.), Lowney (G.), Wallace (S.), Reddi (V. J.) et Hazelwood (K.). – Pin : building customized program analysis tools with dynamic instrumentation. *In : ACM SIGPLAN Notices*. ACM, pp. 190–200.
11. Nethercote (N.) et Seward (J.). – Valgrind : a framework for heavyweight dynamic binary instrumentation. *In : ACM Sigplan Notices*. ACM, pp. 89–100.
12. Peterson (P.). – F2py : a tool for connecting fortran and python programs. *International Journal of Computational Science and Engineering*, vol. 4, n4, 2009, pp. 296–305.
13. Shalf (J.), Dosanjh (S.) et Morrison (J.). – Exascale computing technology challenges. *In : High Performance Computing for Computational Science – VECPAR 2010*, pp. 1–25. – Springer, 2011.